Exploring and Optimizing Itanium2™ Cache(s) Performance for Scientific Computing

William Jalby
jalby@prism.uvsq.fr
PRiSM Laboratory
45 Avenue des Etats Unis
78035 Versailles Cedex France

Christophe Lemuet
lemuet@prism.uvsq.fr
PRiSM Laboratory
45 Avenue des Etats Unis
78035 Versailles Cedex France

Abstract

Memory hierarchies are a key component in getting high performance on modern microprocessors. To satisfy the ever increasing demand on data rate access, they are also becoming increasingly complex, Itanium2™'s cache system being a good example of this trend: three levels of caches, non blocking caches, high degree of parallelism (up to four memory access per cycle), sophisticated instructions for supporting prefetch and cache control etc …. Although all of these advanced features promise to offer large performance gains, in many cases, performance remains disappointing. In this paper, we study in detail performance behavior of simple scientific kernels (BLAS1) on an Itanium2™ cache system. We demonstrate, through systematic experimentations that performance can be very sensitive to the memory address stream structure, revealing that the underlying (hidden) cache organization (banking, load queues) has a major impact on performance. We develop specific instruction scheduling techniques allowing to reach excellent and very stable performance levels.

1. Introduction

Memory subsystem performance is essential to today's microprocessors [CS98]. Therefore, computer architects, have spent a large effort in inventing sophisticated mechanisms to improve data access rate (both in terms of latency and bandwidth) [FJ95, CB95]: multilevel caches, non blocking caches, out of order execution, prefetch instructions, etc …..

The good side of the story is that these mechanisms allow to offer excellent peak performance numbers. The bad side of the story is twofold: first, these mechanisms require from codes specific characteristics to reach peak performance [Ba95]: for example, caches need spatial/temporal locality, prefetch instructions require regular access stream. Second, the resulting complexity of
the memory subsystem is very high, one of the reason being that most of modern microprocessors simultaneously use a large number (if not all) of the mechanisms listed above. This last point does not only make design and fabrication difficult and expensive, but also, makes performance very sensitive to the interaction between these mechanisms themselves and the codes.

Previously, in the "old days" of vector machines, a similar trend was observed. To reach decent memory bandwidth, Cray XMP had a very complex memory subsystem organized in banks, sections, subsections, etc… The analysis of performance of even very simple codes was already very difficult [OL85].

In this paper, the Itanium2™ architecture was selected as a target for exploring cache performance. The Itanium2™ has an interesting cache system (multilevel, high degree of parallelism, sophisticated prefetch capabilities….) [IN02] combined with an original processor architecture (using aggressively static information provided by the compiler and speculative execution) [Ha00, SA00].

Due to the already complex nature of the problem, our study is currently restricted to the cache subsystem (excluding memory) and to simple vector codes (BLAS 1 type: Copy, Daxpy) yet fairly representative of memory address streams in scientific computing. The choice of scientific computing as a target application area is motivated by the excellent match between scientific codes (easy to analyze statically) and the Itanium2™ architecture (very well designed for exploiting static information).

Even with this rather limited scope in terms of application codes, our study reveals that performance behavior is rather complex and hard to analyze. Overlooked parameters such as relative positioning of

starting array address vis-à-vis cache line boundaries or page boundaries are shown to have major performance impact. In particular, the banking/interleaving structure of the L2 cache and the load/store queue structure are shown to have a major interaction with address stream, potentially inducing large performance loss.

We propose several techniques for scheduling Loads and Store Instructions, taking into account the bank structure of the L2 and the load/store queue mechanisms. These techniques are applied to the Copy and Daxpy kernels and experimental results validating the interest of such techniques are presented: optimal (or close to) performance levels are obtained and the performance itself does no longer suffer from instabilities due to the relative position of starting array address.

Section 2 describes our experimental setup: hardware platform as well as software platform (compiler and OS). In section 3, our target codes are presented. In Section 4, the experimental methodology is detailed: this covers parameter space description, measurement methodologies and results presentation/visualization. In Section 5, experimental results on simplified kernels are presented allowing a clear isolation of performance problems. In Section 6 (resp. 7), experimental results on Copy (resp. Daxpy) are presented and analyzed. Finally, in Section 8, a few concluding remarks and directions for future work are given.

2. Experimental setup

2.1. Hardware setup

The machine used in our experiments is a uniprocessor Itanium2™ based system equipped with 900Mhz processor and 3GB memory. The "general" processor architecture (an interesting combination of VLIW and advanced speculative mechanisms) is very well described in the literature [Ha00,SA00,Ba00,IN02]. Of particular importance in our study, is the fact that Itanium2™ is mostly an "in order processor": instructions are executed in the same order they are issued making instruction scheduling rather important. The only "major" exception to this rule is memory operation processing for which a partial out of order processing is allowed through the use of Load/Store queues.

The Itanium2™ offers a wide degree of parallelism:

- Six general purpose ALU, two integer units and one shift unit;
- The Data cache unit contains for memory ports allowing to service either four load requests or two load and two store requests in the same cycle;
- Two floating point Multiply Add units allowing to execute up two floating point multiply add operations per cycle;
- Six multimedia functional units;
- Three Branch units, ….

All of the computational units are fully pipelined, so each functional unit can accept one new instruction per clock cycle (in the absence of stalls).
Instructions are grouped together in blocks of three instructions (called a bundle). Up to two bundles can be issued per cycle. Due to the wealth of functional units, a rate of six instructions executed per cycle can be sustained. This has to be compared with the Itanium1™ processor where an important performance limitation was due to a shortage of memory access units. This shortage was very sensitive on memory bound computations.

On our test machine, caches are organized in three levels, all of them being on chip:

- L1 level split in a D cache (16KB) and an I cache (16KB), using 64 Bytes cache lines. However these caches cannot be used for storing floating point data;
- L2 level, unified, 256 KB, 8 way associative, uses a Write Back Allocate policy and a 128 Bytes cache line;
- L3 level, unified, 1.5 MB, 12 way associative, uses a 128 Bytes cache lines.

Latencies and bandwidth (for the floating point load/store instructions) of the various levels are given in the table below:

	Bandwidth (FP)	Latency (FP)
L1D	NA	NA
L2	32Bytes/cycle READ, or 16Bytes/cycle READ and 16Bytes/cycle WRITE	6 cycles minimum
L3	32B / cycles	12 cycles minimum

The L2 is capable of supporting up to four memory accesses per cycle. The L2 cache is organized in 16 banks, with an interleaving of 16Bytes: address 0 and 8 are located in bank 0, address 16 and 24 in bank 1, etc … The L2 cache non blocking nature is supported via a Load/Store queue (L2OzQ) capable of holding up to 32 operations which cannot satisfied by the L1D. This queue also allows making additional requests to the L2 while younger requests are blocked due to bank conflicts for example. In addition to bank conflicts other problems such as disambiguation of memory address have to be dealt with. A sequence of a Load and a Store addressing the same memory location should be detected and proper ordering should be enforced to preserve program

correction. Ideally, such a specific treatment should be triggered only by comparing the full address of Load and Store. Unfortunately if Loads and Stores execution are too close to each other, only a partial comparison (on the lower address bits) is carried out, generating potential performance penalties between memory references which have been improperly disambiguated. Finally, a maximum of 16 outstanding cache miss (request to L3 or memory) to unique cache lines can be handled by the L2 cache controller.

In addition to standard Load and Store instructions on floating point operands (single and double precision), the Itanium™ instruction set offers Load Floating Pair instruction capable of loading 16 Bytes at once, provided that the corresponding address are lined up on a 16 Bytes boundary.

2.2. Software environment

The test machine was running Linux IA-64 Red Hat 7.1 based on the 2.4.18 smp kernel. The page size used by the system was 16Kbytes.
Two different compilers were used:

- Intel® C++ Compiler Version 6.0, Build 20020614;
- Intel® C++ Compiler Version 7.0 Beta, Build 20020703.

The V6 version was used with –tpp2 flag to generate specific code for the Itanium2™.

On both compilers various options have been tested, however for our simple BLAS1 kernels, it was found that the combination of –O3 and –restrict was very powerful, fully using most of the advanced features of Itanium2™ architecture: software pipelining, prefetch instructions, predication, rotating registers… In getting top performance, the "–restrict" option was essential because it allowed the compiler to assume that distinct arrays were pointing to disjoint memory regions, therefore, allowing a full reordering of loads and stores. Fortran compiler was also tested, but again for our simple loops, the code generated was almost identical to the one obtained with C language.

Besides the compiler and OS, the perfmon toolkit, allowing direct access to various performance counters has been used.

3. Target codes

Two different types of kernels were used. The first one (called *memory stress kernels*) corresponds to artificial codes (i.e. they do not perform "useful" computations), the main goal being to explore cache system behavior. The second type corresponds to two typical *BLAS1 kernels* performing real useful computations.
In this paper results for two *memory stress kernels* are presented (identifiers X and Y refer to double precision floating point arrays):

- **Load/Load Load X(I), Load Y(I):** 2 "independent" load streams, no floating point arithmetic operations
- **Load/Store Load X(I), Store Y(I):** 1 load stream, 1 store stream, no floating point arithmetic operations

These two kernels were directly generated by hand in assembly code. Since they only consist of memory access instructions and simple address computation, we used systematically the mmi (or mmb) bundles. The goal of these kernels is first to test the maximum sustainable bandwidth and second to detect performance problems and third to develop workarounds. These kernels have been tested using different instruction ordering, either alternating between X and Y references (Load X(0), Load Y(0), Load X(1), Load Y(1), etc ….) or grouping (vectorizing) X and Y references (Load X(0), Load X(1), Load X(2), Load X(3), Load Y(0), Load Y(1), Load Y(2), Load Y(3) etc ….)

The two *BLAS1 kernels* used in this paper are simple vector loops:

- **Copy:** ***Y(I) = X(I)***, 1 load, 1 store streams ;
- **Daxpy:** ***Y(I) = Y(I) + a * X(I)***, 2 loads, 1 store streams but 1 load and the store stream referring to the same array Y;

For each of these two kernels, several source code variants hand generated (unrolled, vectorized) were developed, compiled and tested. However, only results for the simplest form (called Std) are reported since the compiler was already performing advanced optimizations and the performance gains of the other variants were negligible.

Following our study of the memory stress kernels, specific optimized versions of both Copy and Daxpy were generated by hand, incorporating the specific instruction scheduling techniques, which were found efficient on the memory stress kernels.

4. Measurement methodology

4.1. Parameter space

Besides the different kernels (described in the previous section) and their variants, other "major" parameters have been explored:

- **Operand location:** since we were interested in exploring cache performance (L2 and L3), two types of measurements were performed: the first one (called L2 region) in which all of the operands are located in L2, the second one (called L3 region) in which all of the operands are located in L3. This "localization" of the operands is achieved by a specific organization of the "*drive*r" code described later in Section 4.2.
- **Prefetch distance and mode:** for each of the loop variant, the compiler selected specific prefetch instruction type and distance. To analyze the impact of prefetch instruction type and prefetch distance, we modified directly in the assembly code the values chosen by the compiler.
- **Starting address of arrays:** in our experiments, the array layout in the virtual memory space is tightly controlled. In particular, the impact of the starting address of each array (X, Y) is studied in depth. To achieve this goal, the parameters Offset X (resp. Offset Y) are introduced, according to the following relations:

Element	Virtual address (Bytes)
X[i]	512K + Offset X + 8*i
Y[i]	512K + DIST1 + Offset Y + 8*i

DIST1 is an additional parameter mainly used to avoid array overlap, i.e. making sure that array X and Y are located in disjoint portions of memory space. In most of our experiments DIST1 remains constant, equal to 32 KB.

It could be argued that the starting address of arrays does not have a major impact on performance, however, as we will see, this is far from being true.

Systematic exploration of this parameter space is fairly expensive: with 2 arrays X and Y, offsets X and Y should both vary between 0 and 16K (the page size) with increment of 8 Bytes (corresponding to 2K values for offset X and 2K values for offset Y). All possible combinations between offset X and offset Y should be tested, leading to 2K x 2K = four millions of experiments!! To limit the combinatorial explosion in terms of experiments, the parameter space corresponding to offsets has been explored in two manners: first experiments were run with offsets varying between 0 and 512 with increments of 8 (spatial 2D exploration), second experiments were conducted with only one offset varying between 0 and 16KB, the other two being set to 0 (1D exploration). It should be noted that such a method could prevent us from discovering all of the pathological behavior. However all of the problems depending upon the relative value Offset X – Offset Y should be captured.

The term "*iteration count*" could become ambiguous, since in some variants unrolling is used. To avoid this problem, the term "*computation size*" is used where "*computation size*" denotes the total number of distinct elements of the array X accessed during the whole loop execution. Impact of the "*computation size*" was not directly studied: for example all of the problems arising with very short "*computation size*" were not tackled.

In our study, we mainly focused on steady state behavior, using a typical "*computation size*" of at least 1440 (corresponding to the computation of 1440 elements) which is large enough for reaching peak performance while still allowing us to keep the operands in the L2 cache.

Across the different variants and kernels, this "*computation size*" remains constant to allow a fair comparison.

4.2. Measurements

Measurements were performed on a stand-alone system (i.e. our benchmark code was the only user application running), only one measurement being performed at a time.

All of the timing measurements were performed using the *mov ar.itc* instruction to read the cycle counter of the processor itself.

Three different types of measurements were used:

1. Mes 1: a standard repetition loop surrounds the code/kernel to be tested. By using an appropriate array size, operands can be "kept" either in L2 or in L3; such a technique generate the well known curves with different plateaux corresponding to the different cache levels.
2. Mes 2: only one execution of the kernel is performed. For the L2 region measurements, the arrays are first loaded in the L2, and then the kernel is executed once. For the L3 measurements, the arrays are flushed from L2 (without being flushed from L3) before kernel execution.
3. Mes 3: several executions of the kernel are performed. In L2 region, the method is equivalent to Mes 1 while in L3 region, the Mes 3 method performs an "auto" flush of the arrays, by moving

through the array through the consecutive executions of the kernel.

While Mes1 and Mes3 modes can accommodate relatively low accuracy timers due to the repetition, Mes2 mode requires a fairly accurate timer.

For all of our experiments, the three types of measurement were performed and consistency between the three results obtained was systematically checked.

In addition, perfmon toolkit reading the various cache miss counters were used first to verify our assumptions that operands were effectively kept in the desired cache level and second that the penalties associated with DTLB remained negligible.

4.3. Results presentation

All of the performance numbers presented are normalized with respect to one iteration: i.e. the measurements correspond:

- To the average number of cycles to perform one pair of memory access: Load X(I) and Load Y(I) (resp Load X(I) and Store Y(I)) for the memory stress kernels;
- to the "average" number of cycles to compute one element of the result in the case of the BLAS1 kernels.

One of the major points of focus will be the impact of offsets on performance. Therefore, 2D plots (isosurface) will be systematically displayed. A "geographic" color code is used: dark blue corresponds to the best performance (lowest number of cycles) while dark red corresponds to the worst performance. These 2D plots will be very useful in understanding qualitatively the spatial nature of the phenomenon. Also, standard curves corresponding to cuts through the 2D plots will be presented to give a precise quantitative measure. The values for these cuts correspond to fixed values of offset Y. These values (offset Y = 0 and 392) were somewhat picked up arbitrarily, the main goal of these "cuts" is to provide precise performance numbers.

5. Memory stress kernels

For all the L2 region (resp. L3 region) 2D plots (figures 1 and 2) the same scale (and color code) is used. Similarly, the same vertical axis scale is used for the L2 region (resp. L3 region) curves.

5.1. Load Load results (Figure 1)

All of the kernels were unrolled 8 times first to minimize the impact of branch penalty and second to minimize the relative cost of prefetch instructions. Two types of Load/Load kernels were used, the first one called interleaved, the second one called vector optimized.

5.1.1. Load/Load Interleaved results

The corresponding interleaved version in assembly code is given below:

```
.b1_2:
 { .mmi
    ldfd    f32=[r32],64  // load x[8*i]
    ldfd    f72=[r33],64  // load y[8*i]
    nop.i   0
 }{ .mmi
    ldfd    f37=[r20],64  // load x[8*i+1]
    ldfd    f77=[r2 ],64  // load y[8*i+1]
    nop.i   0 ;;
 }
            ...
 {  .mmi
    ldfd    f62=[r25],64  // load x[8*i+6]
    ldfd    f102=[r18],64 // load y[8*i+6]
    nop.i   0
 }{ .mmb
    ldfd    f67=[r26],64  // load x[8*i+7]
    ldfd    f107=[r19],64 // load y[8*i+7]
    br.ctop.sptk    .b1_2 ;;
 }
```

Stop bits have been inserted after every group of two bundles, therefore the peak performance of 4 loads per cycle (0.5 cycle per "iteration", an iteration corresponding to the access of one element of X and one element of Y) should be reachable. A variant without any stop bits was also generated and tested, leading to the same results as the one given above.

The performance figure when operands are located in L2 is given in Figures 1a and 1b. Basically, two types of phenomena can be observed:

- Three diagonals separated by 256 Bytes, along which performance goes up to 1 cycle;
- A grid type pattern (sometimes overwritten by the diagonal patterns): on the "good" points (Offset X and Offset Y being of the form 16r + 8), performance is optimal (0.5 cycles) while on the other points performance degrades up to 0.9 cycles.

Both phenomena can be attributed to bank conflicts generated by the interaction between the L2 interleaving scheme and the address streams. The table below summarizes the main bank conflicts occurring with the interleaved Load Load variant. In this table only the block of the first 4 accesses is displayed, the other ones being easily obtained from this initial pattern.

Offset Y values	Offset X values	Load X(0), Load Y(0), Load X(1), Load Y(1) L2 Bank number accessed	Performance (cycles per Load/Load)
Offset Y = 0	Offset X = 0	0 0 0 0 (quadruple conflict on bank 0)	1
Offset Y = 0	Offset X = 8	0 0 1 0 (triple conflict on bank 0)	1
Offset Y = 0	Offset X = 64	8 0 8 0 (two double conflicts on bank 0 and 8)	0.9
Offset Y = 0	Offset X = 72	8 0 9 0 (double conflict on bank 0)	0.9
Offset Y = 8	Offset X = 0	0 0 0 1 (triple conflict on bank 0)	1
Offset Y = 8	Offset X = 8	0 0 1 1 (two double conflicts on bank 0 and 1)	0.9
Offset Y = 8	Offset X = 64	8 0 8 1 (double conflict on bank 8)	0.9
Offset Y = 8	Offset X = 72	8 0 9 1 (no conflict!!)	0.5

The 256 Bytes period of the diagonal patterns stems from the fact bank allocation is periodic with a 256 Bytes period: i.e. two elements of the same array, which are 256 Bytes apart, are allocated to the same cache bank.

As a main conclusion, the classic interleaved access is generating numerous bank conflicts leading to very unstable performance.

5.1.2. Vector Optimized Load/Load

The basic idea is to build a memory access pattern, such that no bank conflict occurs. Therefore, the technique is first to vectorize memory access on X and Y (grouping together 8 consecutive accesses on X then 8 consecutive accesses on Y) and then sorting them in an odd-even manner. The corresponding assembly code is given below:

```
.b1_2:
 {   .mmi
   ldfd    f32=[r32],64   // load x[8*i]
   ldfd    f37=[r21],64   // load x[8*i+2]
   nop.i   0
 }
 {   .mmi
   ldfd    f52=[r23],64   // load x[8*i+4]
   ldfd    f57=[r25],64   // load x[8*i+6]
   nop.i   0 ;;
 }
 {   .mmi
   ldfd    f42=[r20],64   // load x[8*i+1]
   ldfd    f47=[r22],64   // load x[8*i+3]
   nop.i   0
 }
 {   .mmi
   ldfd    f62=[r24],64  // load x[8*i+5]
   ldfd    f67=[r26],64  // load x[8*i+7]
   nop.i   0 ;;
 }
   ...
Loads on y[0], y[2], y[4], y[6], y[1], y[3]

{   .mmb
   ldfd    f102=[r17],64 // load y[8*i+5]
   ldfd    f107=[r19],64 // load y[8*i+7]
   br.ctop.sptk    .b1_2 ;;
 }
```

With such a scheduling of loads, every group of four loads (delimited by stop bits) necessarily address four distinct banks. The use of stop bits is important to preserve this specific grouping.

The resulting performance numbers are given on Figure 1c and 1d, the performance is optimal, (i.e. 0.5 cycle for a Load/Load pair) and flat: all of the bank conflicts have been eliminated.

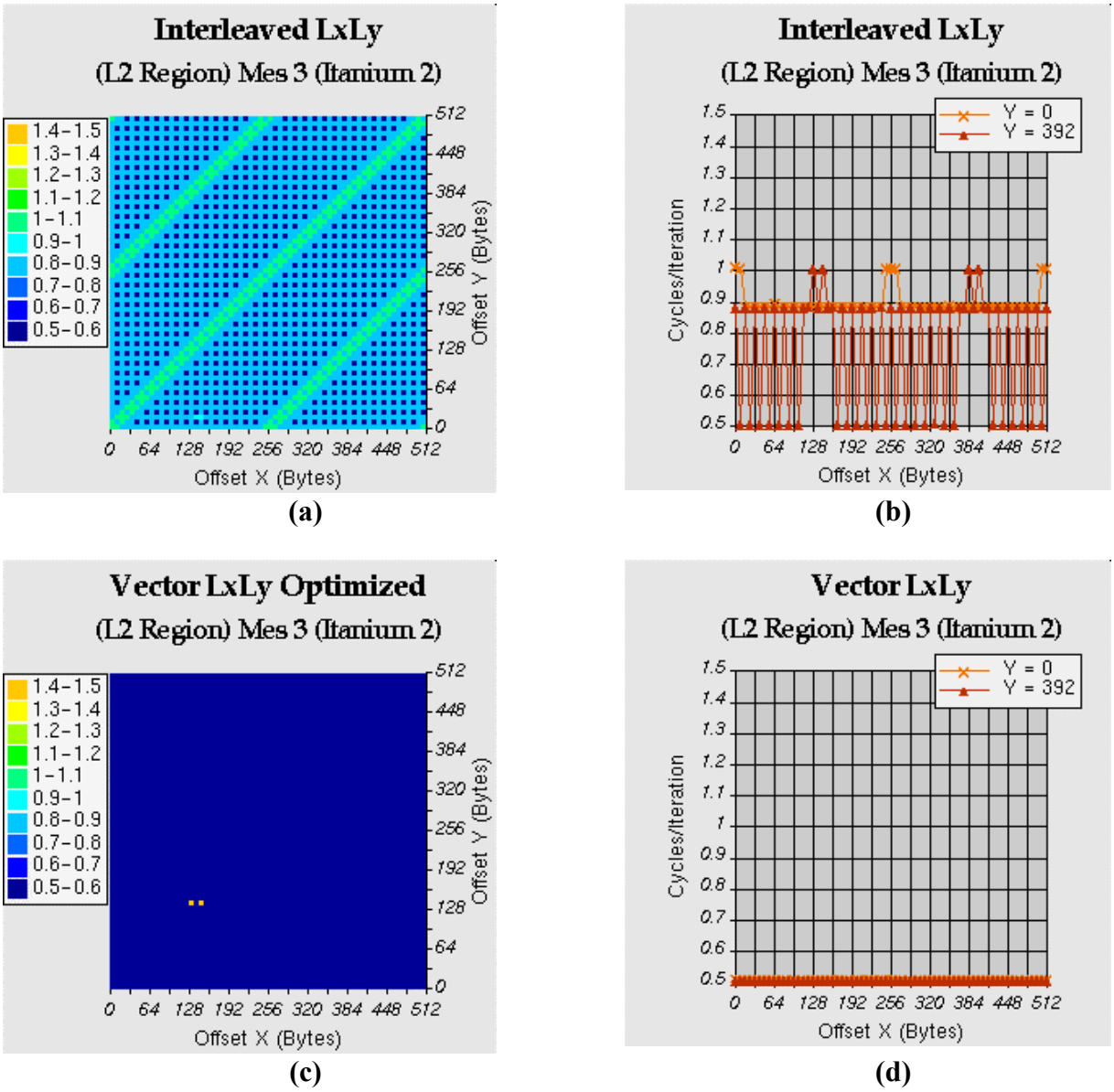


Figure 1. Interleaved LxLy and Vector LxLy Optimized (Itanium 2™).

5.2 Load Store results (Figure 2)

Again two variants have been tested: interleaved and another variant called Vector Optimized.

5.2.1 Load Store Interleaved results

The Load/Store interleaved memory access pattern is very similar to the Load/Load interleaved, the second Load on Y being replaced by a Store.

The corresponding performance results are displayed in Figure 2a and 2b. Although they present some similarities with the Load/Load interleaved variant, there are a number of differences.

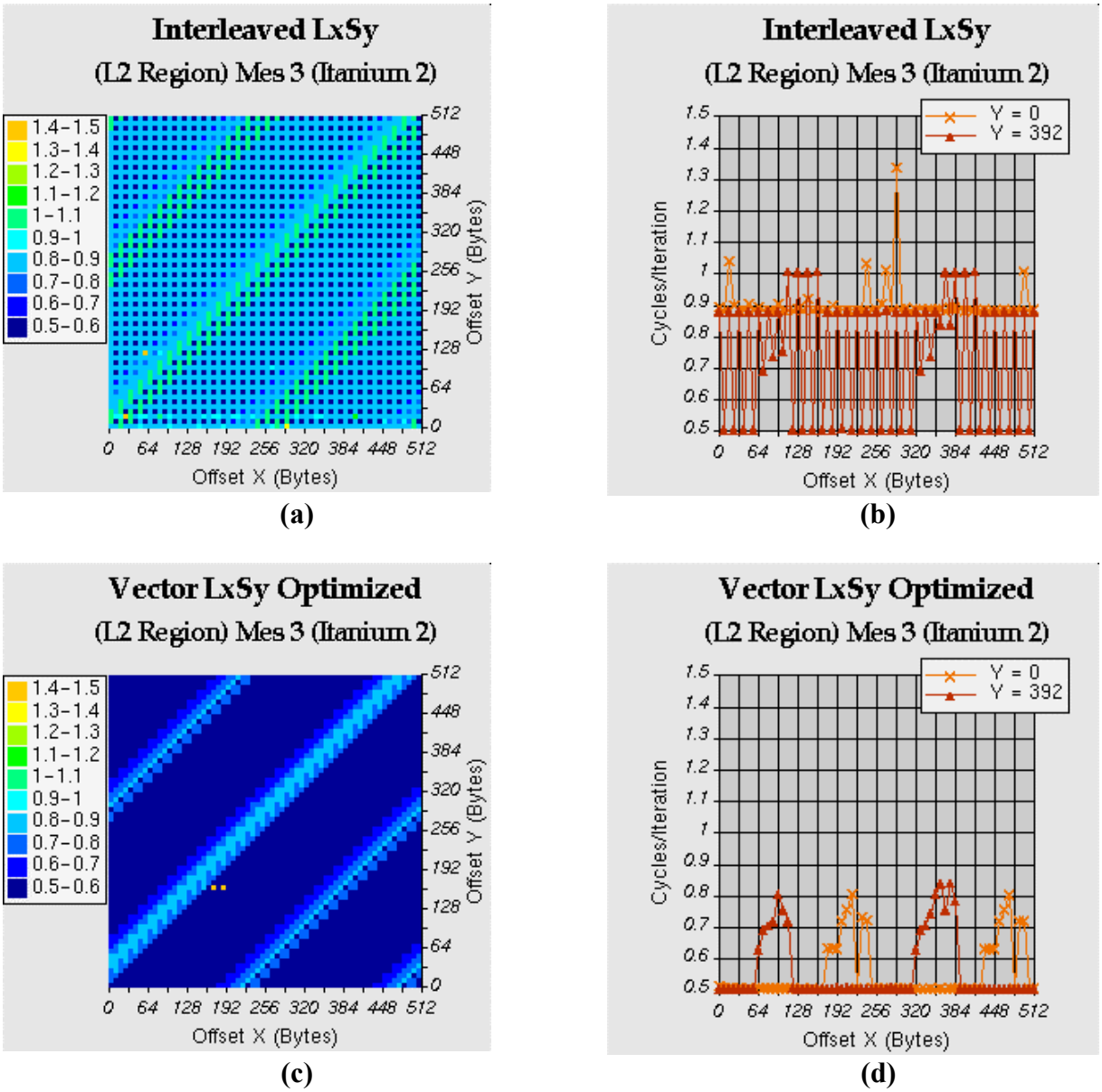


Figure 2. Interleaved LxSy and Vector LxSy Optimized (Itanium 2™).

- A grid pattern can be clearly observed. Again, it is due to bank conflicts but it is more complex to analyse than the previous because a Store will only conflict at the bank with Stores that are issued three cycles later.
- Three diagonal patterns can still be observed but the main diagonal one is higher around 1.3 cycles. This is due to the interaction between a Load and a Store, which have exactly the same low order 16 bits, therefore causing a disambiguation problem because they are issued in the same cycle.

5.2.2. Vector Optimized results

The strategy used for the Load/Load can no longer be applied here because grouping stores by blocks of 4 stores per cycle hits a key performance barrier: a maximum of two stores can be issued per cycle.

Therefore the technique used is group Load and Stores in the following manner:

Load X(0), Load X(2), Store Y(0), Store Y(2) ;;

Load X(1), Load X(3), Store Y(1), Store Y(3) ;;
and so on ….

This scheduling technique will not completely eliminate all of the pathological behaviour. It will suppress the grid pattern and part of the Load/Store disambiguation problem. However, the "bad" performance zones will be clearly defined and in fact, they could be easily derived from the memory access pattern: they will correspond to narrow "diagonal" zones. This is confirmed by the experimental results in Figures 2c and 2d. On these figures, it can be observed that performance is optimal (0.5 cycles per Load and Store, i.e. 2 Loads and 2 Stores are executed every cycle) except on diagonals zones 256B apart. The grid pattern has disappeared. To fully get rid of the diagonal zones, an additional technique based on software pipelining will have to be used. This will be detailed in the Copy kernel section. In essence, software pipelining will allow to move the diagonal and by combining two versions, all of the conflicts can be eliminate

6. Copy kernel results

To allow a fair comparison, the same scale is used in the L2 (resp. L3 region) in Figures 3a, 3b, 5a and 5b (resp. 3c, 3d, 5c and 5d).

6.1. Copy Std results (Figure 3)

For this kernel, the Compiler V6.0 was used because the V7.0 compiler recognized the Copy pattern and generated a call to a library, which turned out to be slower than the code generated by the V6.0.

The assembly code generated by the V6.0 compiler is the following:

```
.b1_3:
 {    .mmi
  // update prefetch distance
 (p16) add r32=16,r34
  // prefetch on X and Y arrays
alternately
 (p16) lfetch.nt1       [r34]
       nop.i   0
 }
 {    .mmb
  // store y[i-6]
(p22) stfd    [r3]=f38,8
  // load  x[i]
(p16) ldfd    f32=[r2],8
      br.ctop.sptk    .b1_3 ;;
 }
```

The code is far from being optimal: in particular, too many prefetch instructions are generated, one every iteration alternating between X and Y arrays. Still using such a code structure, performance levels are expected to be 1 cycle per iteration (assuming operands are located in L2).

Unfortunately, as displayed on Figures 3a and 3b, performance is at best of 1.3 cycles per iteration. Furthermore several diagonals patterns appear with performance varying between 1.6 and 2 cycles. Theses diagonal patterns are partly due to bank conflict but also to the lack of precise disambiguation (Load/Store queue problem).

The situation in L3 (Figures 3c and 3d) is very similar to the L2: best performance is around 1.7 cycles while performance along diagonal patterns varies between 2 and 3 cycles. It should be noted the presence of a wider main diagonal zone due to the interaction between Loads and Stores which are not correctly disambiguated. This phenomenon is more visible in the L3 region than in the L2 region. Another interesting phenomenon is related to the wide diagonal zone located in the upper left corner: the performance degradation occurring in this area is due to the interaction between the prefetch instruction (which behaves like a Load) and the Store instruction.

6.2. Copy Optimized results (Figure 4 and 5)

Our Copy optimized code contains two variants (variant1 and variant2), which are called depending upon the value of offset X and Y.

The variant1 is built along the same principles as the Load/Store Vector kernel. First the loop is unrolled 8 times and then the loop is software pipelined with a depth 6: load X(I) is performed at the same as Store Y(I-6). The value of 6 was chosen to accommodate the L2 Load latency. In L2 region, the performance is given in Figure 4a and 4b. As expected, performance is flat and close to optimal (around 0.6 cycles) except on a few diagonals, which are 256 Bytes apart..

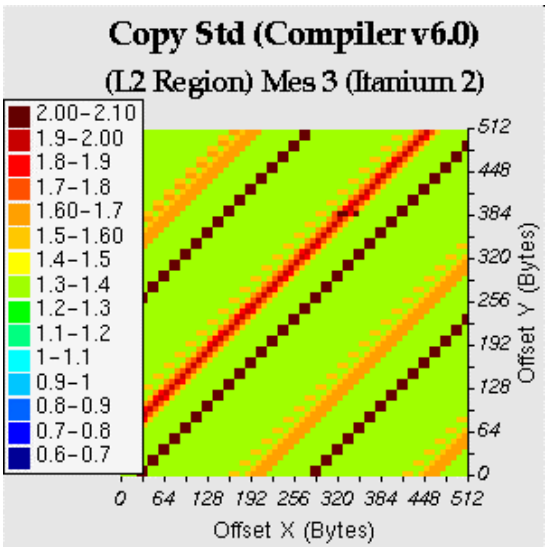
The discrepancy between 0.6 and 0.5 on the Load/Store kernel is due to the presence of prefetch instruction.

Generating variant2 is straightforward, software pipelining is pushed further i.e. at a degree of 22 (6 + 16) : Load X(I) is performed at the same time as Store Y(I-22). Such an increase in software pipeline degree is in fact equivalent to add 128 Bytes (16 x 8) to the offset X value. Now the "bad diagonal zones" have been shifted by 128 Bytes (cf. figures 4c and 4d).

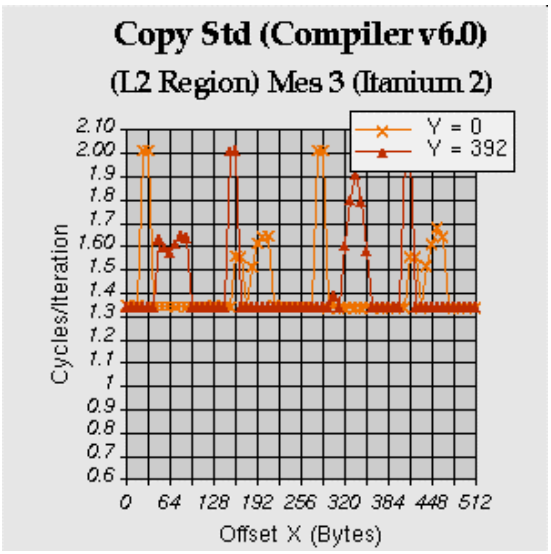
Now, it is easy to combine both variants in the same code, inserting a switch, which will choose the right variant depending upon offset X and Y values. This

generates our final code whose performance is perfectly flat (cf. Figures 5a and 5b).

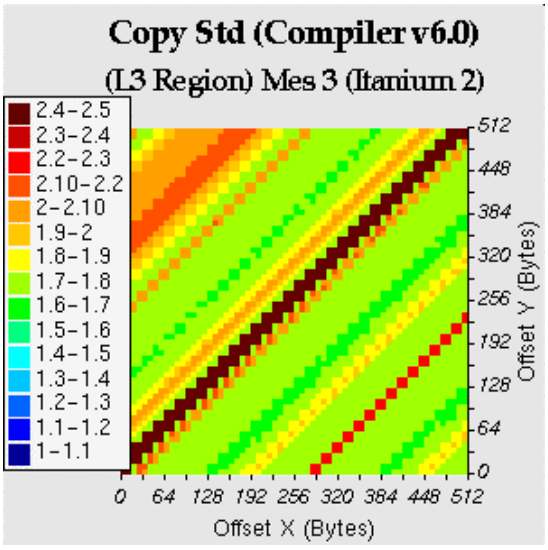
In L3 region (cf. Figures 5c and 5d) , the performance of our variant is flat and better than the compiled V6.0.



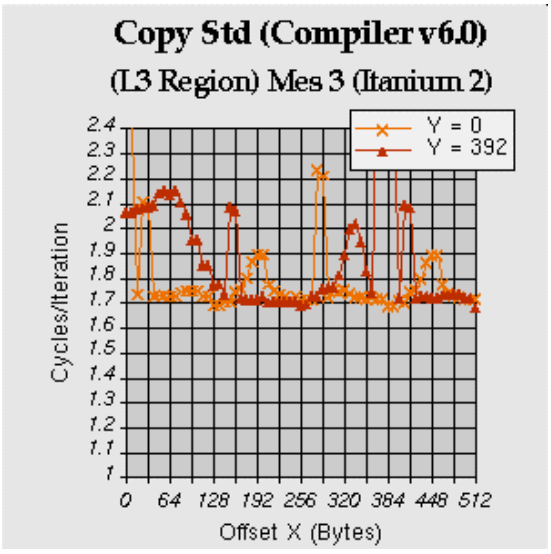
(a)



(b)

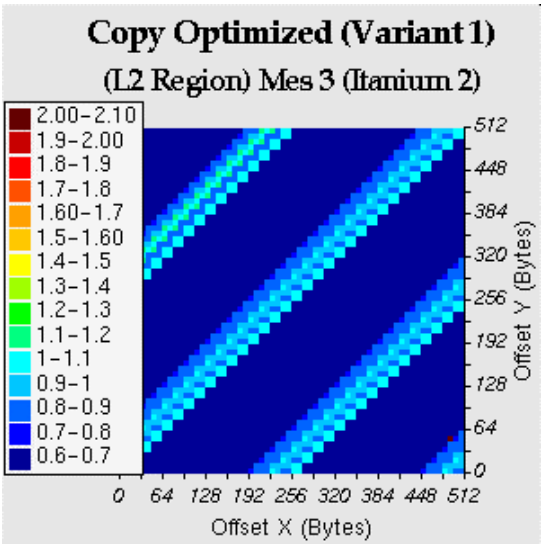


(c)

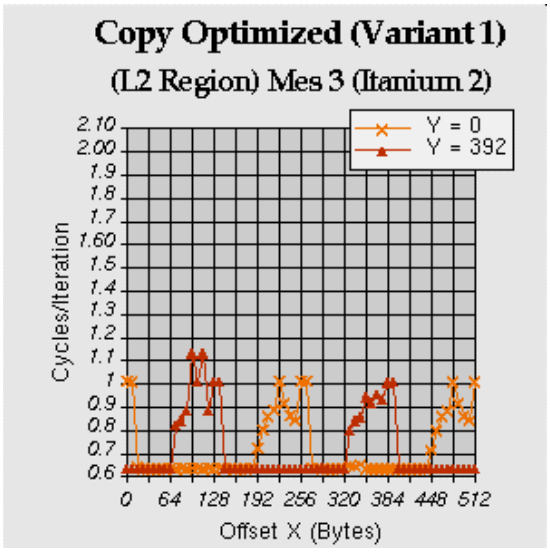


(d)

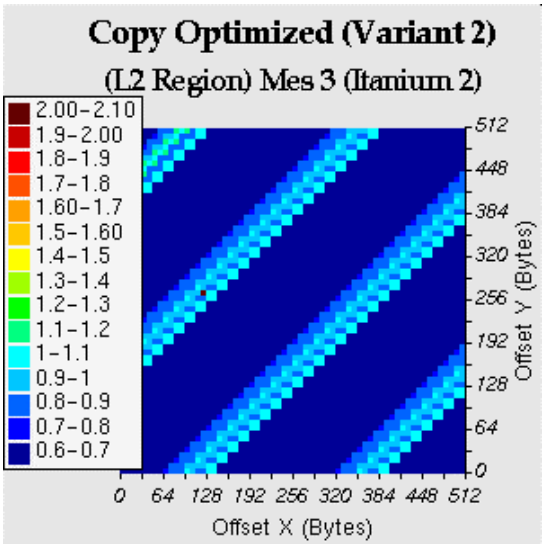
Figure 3. Copy Std (Compiler V6.0) (Itanium2™).



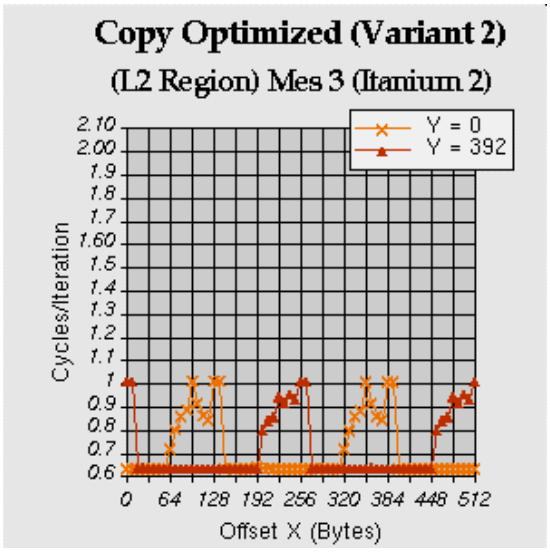
(a)



(b)

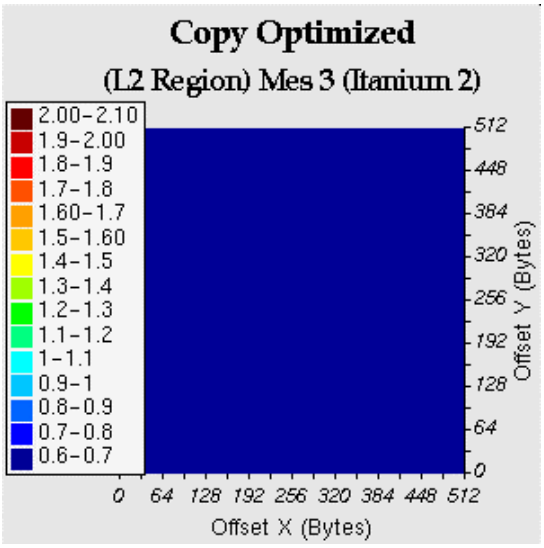


(c)

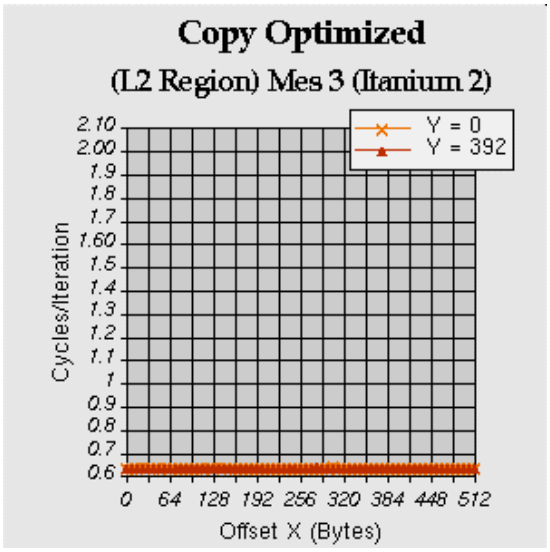


(d)

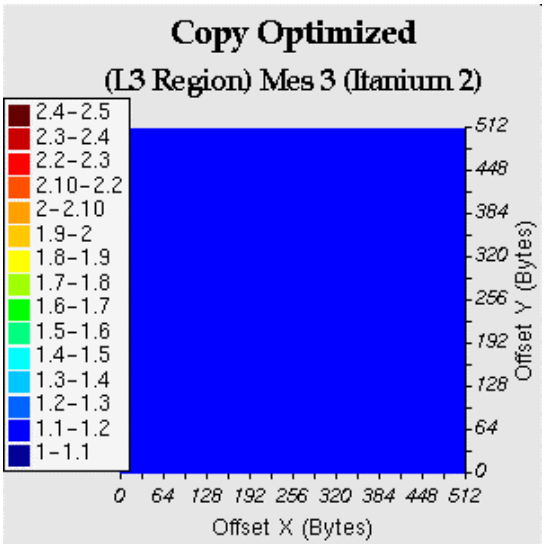
Figure 4. Copy Optimized (Variants) (Itanium2™).



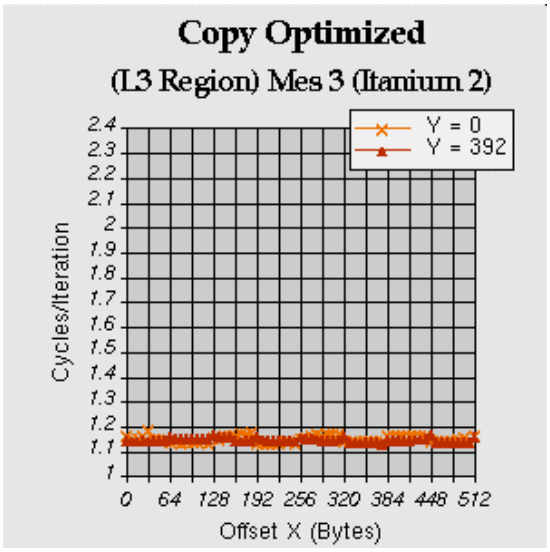
(a)



(b)



(c)



(d)

Figure 5. Copy Optimized (Itanium2™).

7. Daxpy kernel results

To allow a fair comparison, the same scale is used in the L2 (resp. L3 region) in Figures 6a, 6b, 7a and 7b (resp. 6c, 6d, 7c and 7d).

7.1. Daxpy Std results (Figure 6)

The code generated by the V7.0 compiler is fairly complex. It contains three variants, which are used depending upon loop length and Offset y values:

- Variant 1 corresponds to a code unrolled 8 times and uses Load Floating Point pair instructions on Y. Therefore this variant is only used when offset Y is a multiple of 16 Bytes;
- Variant 2 corresponds to a code also unrolled 8 times but without the use of Load Floating Pair

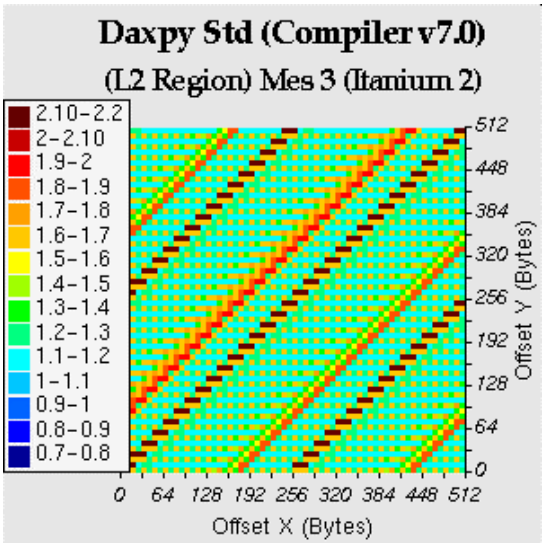
instructions. Therefore it is used when offset Y is not a multiple of 16 Bytes;

- Variant 3 is not unrolled and seems to be used for loop count.

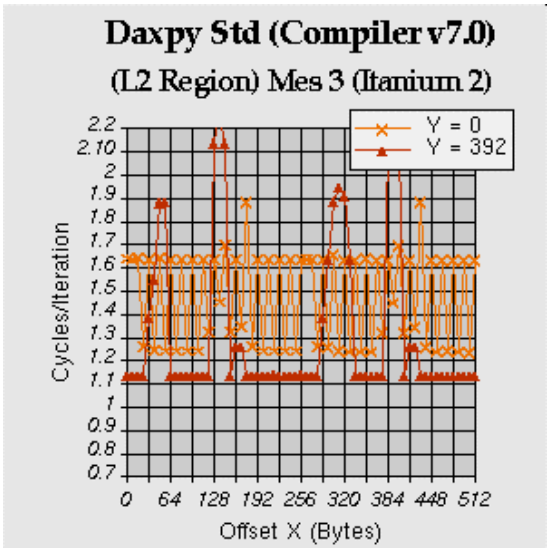
For the three variants, prefetch instructions on X and Y are inserted.

In L2 region (Figures 6a and 6b), performance is oscillating a lot. First a grid pattern can be observed somewhat similar to the one observed on Load Load Interleaved and Load Store Interleaved kernels. A detailed analysis of the code reveals that this grid pattern is clearly generated by bank conflicts: reference to arrays X and Y coexist in the same bundle. Again most of the diagonals patterns could also be attributed to bank conflicts.

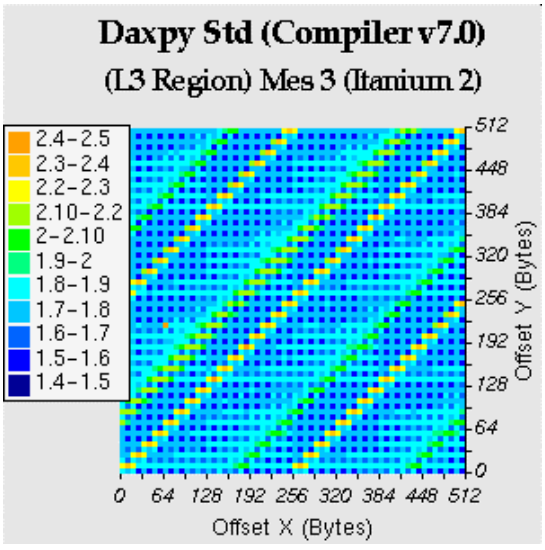
In L3 region (Figures 6c and 6d), the behavior is very similar to the one observed in L2 region. The relative amplitude of the oscillations is slightly reduced.



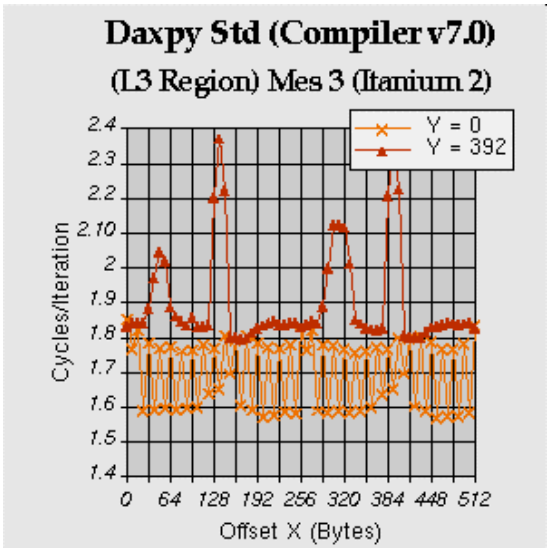
(a)



(b)



(c)



(d)

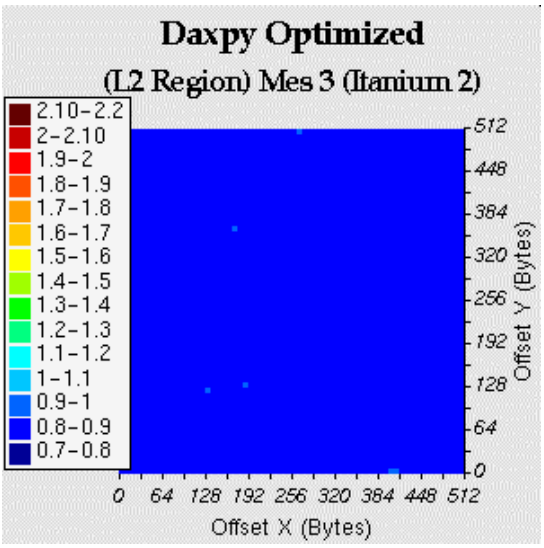
Figure 6. Daxpy Std (Compiler V7.0) (Itanium2™).

7.2. Daxpy Optimized results (Figure 7)

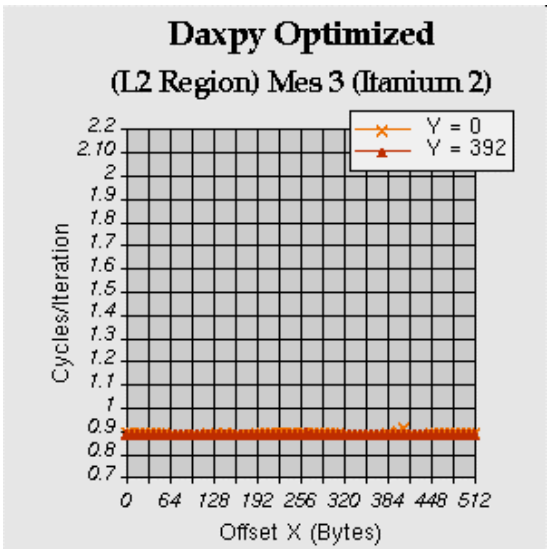
The Daxpy optimized code is obtained by combing the copy memory access pattern (Load on X and Store on Y) and the Load/Load access pattern (Load on Y).
As for the Copy, two variants are necessary to get rid of all of the diagonal zones.

The performance results are presented in Figures 7a and 7b (L2 region) and in Figures 7c and 7d (L3 region). In both cases performance is always better than the V7 compiler.

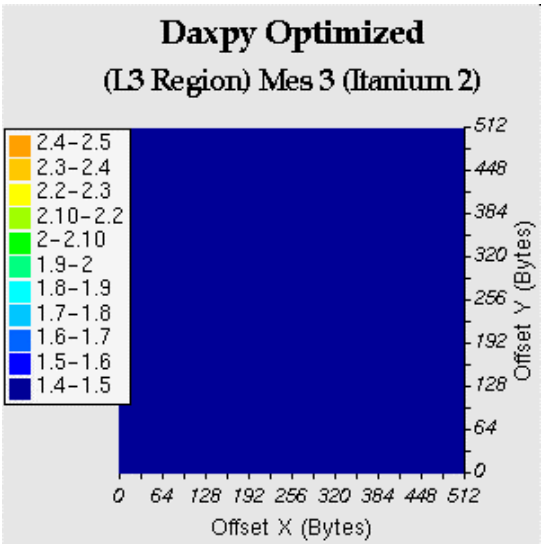
The performance of 0.9 cycles per iteration might seem disappointing while a simple count would lead to 0.75 cycles because one iteration requires two loads and one store (each of them costing 0.25 cycles). However taking into account necessary prefetch instructions, performance at best is 0.82 cycles. Now our optimized version was only unrolled 8 times and therefore, unnecessary prefetch instructions were inserted. Unrolling 16 times would lead to the optimal performance.



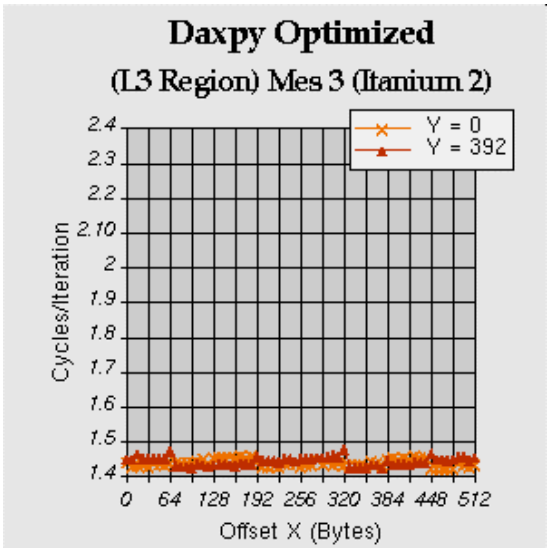
(a)



(b)



(c)



(d)

Figure 7. Daxpy Optimized (Itanium2™).

8. Conclusion

Modern microprocessors rely on complex cache systems to deliver top performance. The dark side of the coin is the resulting complexity, not only for designing them but also for exploiting them efficiently even on simple codes.

Our studies of simple BLAS1 kernels on the Itanium2™ have clearly shown the complex performance behavior of the cache system (both L2 and L3). It was clearly demonstrated that the banking structure of the L2 has a major impact on performance. This banking structure has to be taken into account when scheduling memory access instructions. We explored two specific memory access patterns (Load/Load) and (Load/Store) from which we derived efficient instruction scheduling techniques (variants around vectorization of memory access). These techniques were then applied to two standard BLAS1 primitives and demonstrated stable and close to optimal performance in particular when operands are located in L2.

This work will be extended into four major directions:

- The instruction scheduling techniques described here should be studied in more depth. Vectorization and increasing software pipeline depth increases a lot register pressure. Our Daxpy Optimized variant is close to using 80 floating-point registers!! Therefore, register consumption should be studied in more detailed manner and in particular when dealing with more complex loop bodies, involving for example a larger number of arrays.
- Main memory access deserves a similar study. Already, some preliminary tests have confirmed us with the good performance capabilities of the vectorization strategy;
- New generation of Itanium™ processors are coming up (Madison), which deserve similar studies;

Finally, other processor architectures (Power, UltraSparc, …) should be studied in a similar manner. We already performed a similar set of experiments on Alpha 21264 and Power4: similar phenomena not related directly to banking but to insufficient Load/Store queue disambiguation mechanisms were encountered. Other specific instruction scheduling techniques have to be developed to overcome these weaknesses.

Acknowledgements

This study has been funded by the French Ministry of Research and by Bull. We would like to thank Gerard Roucairol Bull CTO who has triggered our interest for IA64, Jean Papadopoulo Senior Architect at Bull who has strongly supported our effort and finally Bull's IA64 team who has provided us with a timely access to an Itanium2™ platform. We also thank Sid Touati for his careful proofreading of the manuscript.

Bibliography

[Ba95] "Unfavorable Strides in Cache Memory Systems", D.H. Bailey, Scientific Programming, vol. 4 (1995), pp. 53-58

[Ba00] "The Intel IA-64 Compiler Code Generator", Jay Bharadwaj et al, IEEE Micro, Sept -Oct. 2000.

[CB95] "Effective hardware-based data prefetching for high-performance processors", Tien-Fu Chen and Jean-Loup Baer, IEEE transactions on computers, May 1995, 44, 5, pp609-623

[CS98] "Parallel Computer Architecture a Hardware/ Software approach", David~E. Culler, Jaswinder~Pal Singh, and Anoop Gupta. Morgan Kaufman, 1998.

[FJ95] "How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors", Keith I. Farkas, Norman P. Jouppi, Paul Chow, Proceedings of the International Conference on High Performance Computing Architecture, 1995

[Ha00] "Introducing the IA-64 Architecture", Jerry Huck et al., IEEE Micro, Sept - Oct. 2000.

[IN01] "Intel ® Itanium ™ Processor Reference Manual for Software Optimization", Intel Document Number: 245473-003, Nov 2001.

[IN02] "Intel ® Itanium2 ™ Processor Reference Manual for Software Development Optimization", Intel Document Number: 251110-001, June 2002

[JF01] "Application Optimization for Itanium ™ - based Systems", J. Baron, J. Fier, J.P. Panziera, A. Raefsky, Intel Development Forum, Aug 2001.

[OL85] "On the Effective Bandwidth of Interleaved Memories in Vector Systems," W. Oed and O. Lange, IEEE Transactions on Computers, C-34(10):949--957, October 1985.

[SA00] "Itanium Processor Microarchitecuture", Harsh Sharangpani, Ken Arora, IEEE Micro, Sept - Oct. 2000.